Development Approach

Step 1 : Developing dummy kernel stubs, for open(), read(), write() and close() system calls, and dummy implementation of init() and exit() methods.

Step 2 : Implementing the logic for enqueue and dequeue of a circle buffer in the user space. Unit testing of the enqueue and dequeue logic, using scaffolding at the user level

Step 3 : On successfully testing the queue functionality at the user level, I convert the user space symbols to kernel symbol (changing malloc() to kmalloc(); changing free() to kfree()).

Step 4 : Integrating the queue code with the kernel stubs. Re-modelling the init() method code of kernel for creation of device files and initialization of the circle buffer.

Step 5 : Adding code to read data from rtc cmos device.

Step 6 : Developing user application with sender and receiver functionality.

Step 7 : Testing the driver code with 1 sender and 1 receiver for multiple runs.

Step 8 : On successfully testing the driver's functionality with 2 senders and 2 receivers, the application is extended to add additional router functionality.

Test methodology

The circular queue encompasses the crux of this software, since its functionality is critical to the consistency of the system, its testing has been done more thoroughly than the other modules.

Testing the enqueue and dequeue functionality of a circle buffer at the user level with scaffolding :

a) The size of the queue is restricted to a small value like 40 bytes.

b) Since the queue boundary conditions are critical to the vitality of the queue, scaffolding is particularly created keeping the boundary conditions in mind.

The conditions tested are :

1) Queue full : trying to enqueue 5 data packets of size 10 bytes each into a 40 bytes queue.

2) Queue empty : Trying to dequeue 5 data packets of size 10 bytes each in a 40 byte queue.

3) Enqueue during a roll-over : first enqueue two 17 bytes packets into the queue, then perform a dequeue from the head of the queue. Then again try to enqueue at the tail, 6 bytes of data have to be partially added to the end of the queue and remaining 11 bytes after rolling over.

4) Dequeue during a roll-over : from the previous scenario, after the enqueue, I perform 2 consecutive dequeue; the second dequeue operation reads 6 bytes from the end of

the queue and remaining 11 bytes after rolling over.

5) Dequeue during a roll-over when there is less than 4 bytes at queue boundary : This is a special scenario which needs attention. My design dictates that before dequeue the no of bytes to be dequeued has to be read first from the first 4 bytes of the buffer. To test this scenario, I enqueue two 19 bytes packet, then dequeue 1 packet and then add another 19 bytes packet at the end.

Test Case	**Operation (Q_SIZE = 40)**	**Result**
Queue empty	Dequeue() without any items in the queue	Success

Queue full	Packet size = 17
	enqueue(q, packet);
	enqueue(q, packet);
	enqueue(q, packet); // Q is full
		Success

Enqueue during a roll-over	Packet size = 17
	enqueue(q, packet)
	enqueue(q, packet)
	dequeue(q, packet)
	enqueue(q, packet)//Roll over
		Success

Dequeue during a roll-over	Packet size = 17
	enqueue(q, packet) // h =0 t =17
	enqueue(q, packet) // h =0 t =34
	dequeue(q, packet) // h=17 t=34
	enqueue(q, packet) // h=17 t = 11
	dequeue(q, packet) // h =34 t =11
	dequeue(q, packet)// h =11 t=11
		Success

Dequeue during a roll-over when there is less than 4 bytes at queue boundary	Packet size = 19
	enqueue(q, packet) // h =0 t =19
	enqueue(q, packet) // h =0 t =38
	dequeue(q, packet) // h=19 t=38
	enqueue(q, packet) // h=19 t = 17
	dequeue(q, packet)// h =38 t=17
	dequeue(q, packet)// h =17 t=17
		Success

The circular queue is the critical functionality of this development project, hence the maximum testing effort was devoted to testing this functionality; The tests performed on other modules are insignificant compared to these tests.

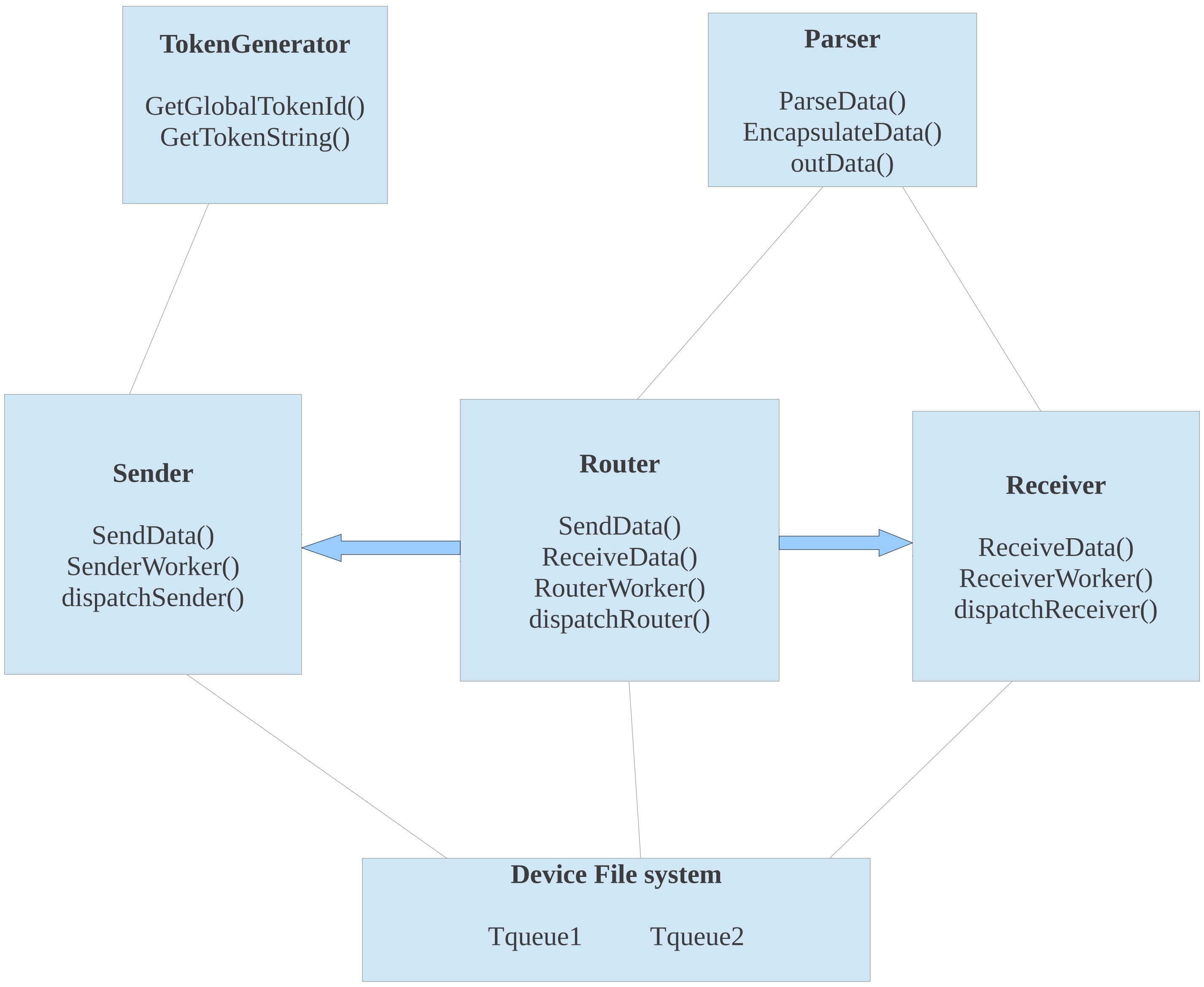
Design Illustration

This development project can be divided into 3 modules :

a) The User module
b) The kernel module
c)The queue module

The design of each module is illustrated below:

The User module : This module in turn consists of 3 sub – module ; Sender, Receiver, Router; These submodules use the uniform device interface provided as device files Tqueue1 and Tqueue2 by the driver (T_Queue_driver) to communicate with the device



Sender

1) dispatchSender() : Entry point for the sender threads.
2) SendData() : Writes data packet to the device
3) senderWorker() : Packs user data (token_id+token_string) into a buffer and invokes sendData() method. It continues the operation for 100 times.

Receiver

1) dispatchReceiver(): Entry point for the receiver threads
2) receiveData() : Reads data packet returned by the device.
3) receiverWorker(): Unpacks the data returned from the driver parses the data into human readable format and writes to the console. It continues the operation for 100 times.

Router

1) dispatchRouter(): Entry point for the router threads

2) SendData() : Writes data packet to the device
3) receiveData() : Reads data packet returned by the device.
4) RouterWorker() : Combines the operations of senderWorker() and receiverWorker(), this operation is performed for 100 times.

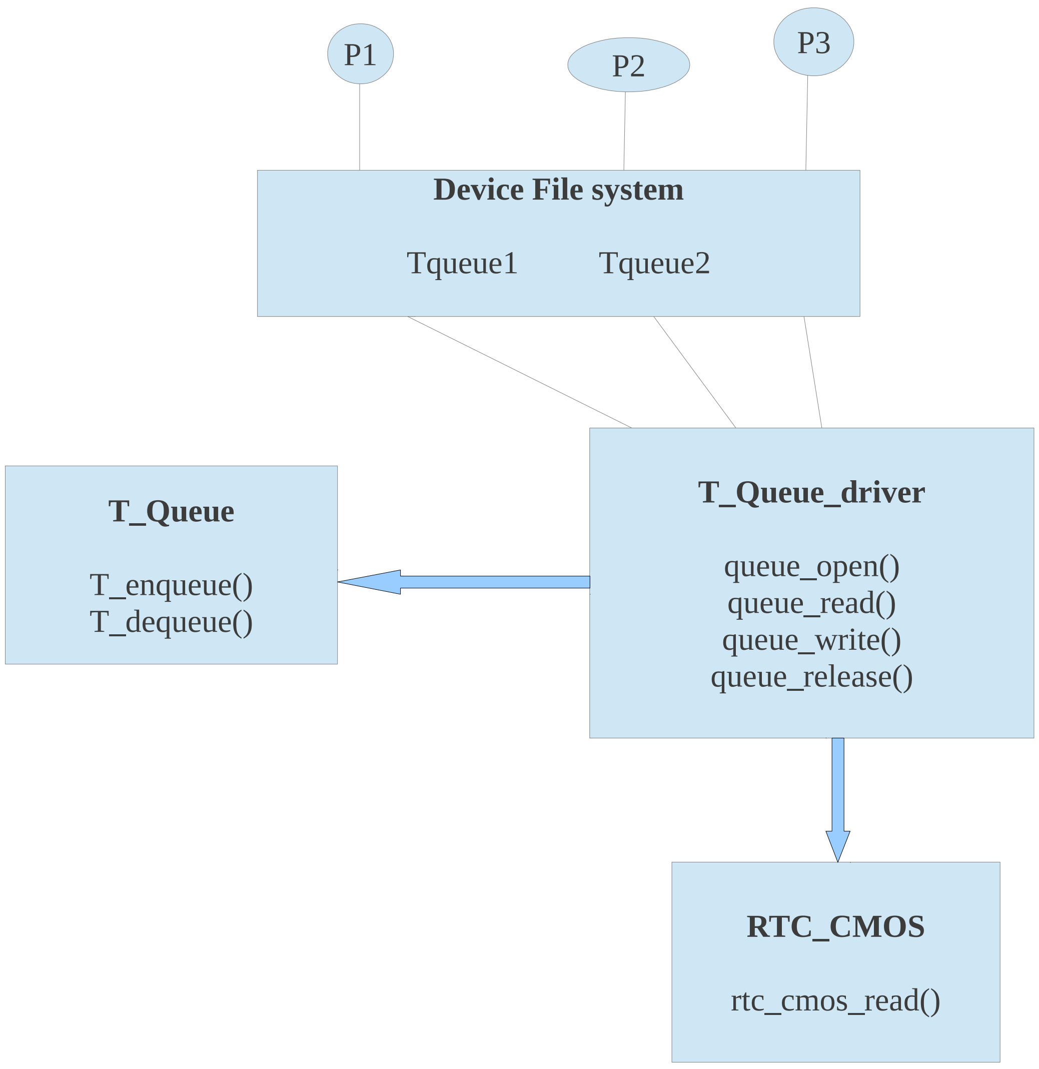
TokenGenerator

1) getGlobalTokenID() : generates a unique token id, this method is serialized
2) getTokenString : generates a random string with a size ob/w 10 to 80.

Parser

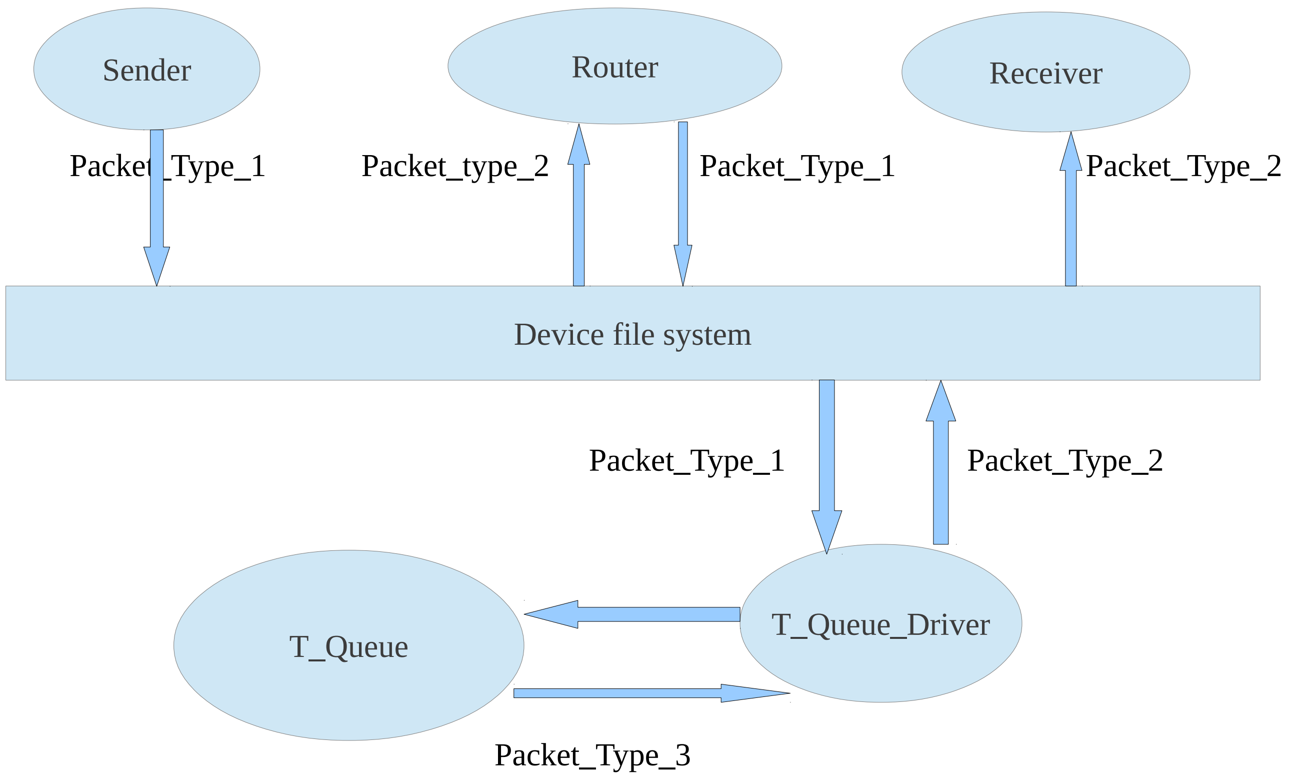
1) encapsulateData() : This methos is used to pack data into a buffer before sending it to the device.
2) ParseData() : Unpacks the data received from the driver and converts to human readable format.
3) OutData() : prints the parsed data to the console.

The kernel module : Implements the open(), read(), write() system calls and performs the necessary operation by invoking the method implemented by the queue module.



Data Flow Model

This model describes the data flow and illustrates the packet structures of the packets being transferred b/w the user module, kernel module and the queue.



Packet_Type_1 :

Token_ID	Token_String

Packet_Type_2 :

Length	Token_ID	Token_String	Arrival_time	Dep_time

Packet_Type_3:

Length	Token_ID	Token_String	Arrival_time

Length : Length of the entire packet considering itself
Arrival_time : Arrival time stamp
Dep_time : Time when the packet leaves the queue.

Performance measurement for the Driver

Performance stats for T_Queue_driver generating 100 tokens and time stamps

```
    16,982,467        cpu-cycles:u          #    0.000 GHz              [71.74%]
    71,217,760        cpu-cycles:k          #    0.000 GHz              [71.71%]
    3,208,262         instructions:u        #   0.07  insns per cycle     [97.50%]
    13,814,393        instructions:k        #   0.31  insns per cycle     [29.26%]
    0                 context-switches:u
    445               context-switches:k
    0                 kmem:kmalloc:u
    908               kmem:kmalloc:k

   0.532496280 seconds time elapsed
```

Performance comparision between T_Queue_driver, message passing and pipes

Message Passing

Performance counter stats for and application performing 500 writes and reads using message passing mechanism

```
    1,381,117         cpu-cycles:u                 #0.000 GHz
    6,308,518         cpu-cycles:k                 #    0.000 GHz
    777,209           instructions:u               #    0.20  insns per cycle
  <not counted>       instructions:k
    0                 context-switches:u
    36                context-switches:k
    0                 kmem:kmalloc:u
    540               kmem:kmalloc:k

   0.056600774 seconds time elapsed
```

Pipes

-

Performance counter stats for an application performing 500 writes and reads using pipes.

```
     1,200,247        cpu-cycles:u          #    0.000 GHz
     6,332,353        cpu-cycles:k          #    0.000 GHz
       677,283        instructions:u        #   0.18  insns per cycle
  <not counted>       instructions:k
             0        context-switches:u
            36        context-switches:k
             0        kmem:kmalloc:u
            39        kmem:kmalloc:k

   0.055611470 seconds time elapsed
```

T_Queue_driver

Performance counter stats for an application performing 500 writes and reads using T_Queue_driver.

```
      1,370,335          cpu-cycles:u             #    0.000 GHz                    [66.92%]
     15,880,600          cpu-cycles:k             #    0.000 GHz                    [62.57%]
        318,817          instructions:u           #    0.04  insns per cycle
      7,214,711          instructions:k           #    0.84  insns per cycle          [44.33%]
              0          context-switches:u
             12          context-switches:k
              0          kmem:kmalloc:u
            817          kmem:kmalloc:k

    0.012706778 seconds time elapsed
```